

Containercomponents

Container Overview ■ Container components are the smart components in the PlayPlus architecture that handle business logic, data management, and state coordination. They act as the bridge between your application's data layer and the presentational components. Key Characteristics ■ Smart Components ■ Handle business logic and data operations Manage application state using Angular Signals Coordinate between services and presentational components Handle loading states, error states, and data transformations State Management ■ Use Angular Signals for reactive state management Implement loading, error, and data states Provide computed values for derived state Handle async operations with proper error handling Data Flow ■ Inject and use services for data operations Transform data for presentational components Handle user actions and events Manage component lifecycle Template Structure ■ TypeScript Component (component.ts.hbs) ■ import { Component , ChangeDetectionStrategy , inject , signal , computed , } from "@angular/core" ; import { CommonModule } from "@angular/common" ; import { takeUntilDestroyed } from "@angular/core/rxjs-interop" ; import { Subject } from "rxjs" ; @ Component ({ selector : "app-example" , standalone : true , imports : [CommonModule] , templateUrl : "./example.component.html" , styleUrls : ["./example.component.scss"] , changeDetection : ChangeDetectionStrategy . OnPush , }) export class ExampleComponent { private destroy\$ = new Subject < void > () ; // Service injections private exampleService = inject (ExampleService) ; // Signal-based state management protected loading = signal (false) ; protected error = signal < Error | null > (null) ; protected data = signal < any [] > ([]) ; // Computed values protected hasData = computed (() => this . data () . length > 0) ; protected canLoadMore = computed (() => ! this . loading () && ! this . error () && this . hasData ()) ; constructor () { this . initializeComponent () ; } private initializeComponent () : void { this . loadData () ; } private loadData () : void { this . loading . set (true) ; this . error . set (null) ; this . exampleService . getData () . pipe (takeUntilDestroyed ()) . subscribe ({ next : (data) => { this . data . set (data) ; this . loading . set (false) ; } , error : (err) => { this . error . set (err) ; this . loading . set (false) ; } , }) ; } protected onAction (action : any) : void { // Handle user actions console . log ("Action triggered:" , action) ; } protected onError (error : Error) : void { this . error . set (error) ; } protected retry () : void { this . loadData () ; } } HTML Template (component.html.hbs) ■ < div class = " example-container " > <!-- Loading State --> @if (loading()) { < div class = " loading-state " role = " status " aria-live = " polite " > < div class = " spinner " aria-hidden = " true " > </ div > < p > Loading... </ p > </ div > } <!-- Error State --> @if (error(); as errorData) { < div class = " error-state " role = " alert " > < h2 > Something went wrong </ h2 > < p > {{ errorData.message }} </ p > < button type = " button " (click) = " retry() " aria-label = " Retry loading data " > Try Again </ button > </ div > } <!-- Content State --> @if (!loading() && !error()) { < div class = " content " > @if (hasData()) { < div class = " data-container " > <!-- Presentational components go here --> < p > Data loaded successfully! </ p > < button type = " button " (click) = "

```

onAction('example') " aria-label = " Perform example action " > Example Action </ button > </ div >
} @else { < div class = " empty-state " > < p > No data available </ p > </ div > } </ div > } </ div >

```

Features ■ **Built-in Features** ■ **Service Integration** Automatic service injection using inject()
function Support for multiple services Proper error handling and loading states State Management
Signal-based reactive state Loading, error, and data states Computed values for derived state
Lifecycle Management Automatic cleanup with takeUntilDestroyed() Proper component
initialization Memory leak prevention Error Handling Comprehensive error states Retry
functionality User-friendly error messages Accessibility ARIA labels and roles Keyboard navigation
support Screen reader friendly Best Practices ■ **Service Usage** // Inject services private
exampleService = inject (ExampleService) ; // Use in data loading this . exampleService .
getData () . pipe (takeUntilDestroyed ()) . subscribe ({ ... }) ; State Management // Define
signals protected loading = signal (false) ; protected error = signal < Error | null > (null) ;
protected data = signal < any [] > ([]) ; // Use computed values protected hasData = computed (
() => this . data () . length > 0) ; Error Handling protected onError (error : Error) : void { this .
error . set (error) ; } protected retry () : void { this . loadData () ; } Usage Examples ■ **Basic**
Container Component ■ # Generate a container component playplus generate container user-list
--services=user This creates: UserListComponent with UserService injection Loading, error, and
data states Proper error handling and retry functionality Advanced Container Component ■ #
Generate with multiple services playplus generate container dashboard
--services=user,product,order This creates: DashboardComponent with multiple service injections
Complex state management Coordinated data loading Integration with Presentational
Components ■ Container components typically: Load and transform data from services Pass data
to presentational components via inputs Handle events from presentational components via
outputs Manage loading and error states for the entire view // Example integration @ Component (
{ ... }) export class UserListComponent { protected users = signal < User [] > ([]) ; //
Pass data to presentational component protected userData = computed (() => this . users () .
map (user => ({ id : user . id , title : user . name , description : user . email , isActive : user .
isActive }))) ; // Handle events from presentational component protected onUserAction (userId :
string) : void { // Handle user action } } Testing ■ Container components include comprehensive
test files with: Service mocking State management testing Error handling verification User
interaction testing // Example test structure describe ("UserListComponent" , () => { let
component : UserListComponent ; let userService : jasmine . SpyObj < UserService > ;
beforeEach (() => { // Setup with mocked services }) ; it ("should load users on init" , () => { //
Test data loading }) ; it ("should handle errors gracefully" , () => { // Test error handling }) ; }) ;
Architecture Benefits ■ **Separation of Concerns** : Business logic separated from presentation
Reusability : Presentational components can be reused across containers **Testability** : Easy to test
business logic in isolation **Maintainability** : Clear data flow and state management **Performance** :
OnPush change detection for optimal performance **Next Steps** ■ After creating a container
component: Implement service methods for your specific use case Add presentational
components to display data Customize loading and error states for your UI Add routing if needed

Write comprehensive tests for all scenarios Developer Checklist ■ Before Creating Container Components: ■ Does the component handle business logic and data management? Are services properly injected using inject()? Are all state updates using Angular Signals? Are loading, error, and data states implemented? Is OnPush change detection enabled? Are async operations handled with error boundaries? Are computed values used for derived state? Is takeUntilDestroyed() used for subscription cleanup? Are user actions delegated to services? Do I have unit tests with mocked services? Is data transformed for presentational components?